Performance Comparison of Uniform NURBS Surface Tessellation on the Pentium® III Processor and the Pentium 4 Processor

Version 2.0

7/00

Order Number: 248609-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Pentium® 4 processor update	7/00
1.0	Original publication of document	9/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

Alan Watt, *3D Computer Graphics 2nd Edition*, Addison Wesley, 1993.

James Foley, Andries van Dam, Steven Feiner, John Hughes, *Computer Graphics: Principles and Practice, 2nd Edition*, Addison Wesley, 1990.

Les Piegl, Wayne Tiller, *The NURBS Book*, Springer-Verlag, 1997.

Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method, Intel Application Note AP-803, Order No: 243637-001.

1 Introduction

The Streaming SIMD Extensions (SSE) for the Intel® architecture (IA) instruction set provide packed, single-precision, floating-point, single-instruction multiple-data (SIMD) operations. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio.

This application note discusses optimization techniques for the tessellation of non-uniform rational B-spline (NURBS) surfaces (a type of parametric surface) for display in real-time applications. NURBS are the standard for describing curves and surfaces in the computer aided design industry. Their use is growing for modeling objects in films, digital art and, most recently, video games.

A parametric surface is defined by a set of basis functions and by a set of discrete points called control points. Points that lie on the parametric surface (surface points) are calculated by evaluating a function that requires as input the control points, the basis functions, and two parametric values. An efficient way of rendering a parametric surface is by breaking the surface into a set of discrete polygons that approximate the surface and then rendering the polygons with a standard polygonal rendering pipeline. The process of breaking the surface into a set of discrete polygons is referred to as *tessellation*.

There are two significant optimization opportunities in the tessellation of NURBS surfaces:

- evaluation of the basis functions
- evaluation of the surface points.

This application note does not discuss optimization techniques for basis function evaluation; it assumes that the basis functions have already been evaluated and are stored in an array in memory. This application note focuses on the evaluation of surface points. First, an overview of NURBS is presented. Then, optimization techniques (including techniques that utilize SSE) are discussed. Next, performance on the Pentium® III processor is compared to the performance on the Pentium 4 processor. Finally, source code samples for all of the optimization techniques are contained in the appendices.

2 NURBS Surfaces

Curved surfaces can often be defined and stored more efficiently with parametric surfaces than with polygonal meshes. Many more polygons than parametric surface patches are needed to approximate a curved surface at a moderate to high level of accuracy. Since many real-world objects are inherently curved, it is not surprising that parametric surfaces (specifically, NURBS) are the standard for modeling real-world objects. Historically, parametric surfaces have been heavily used in CAD applications where real-world objects are modeled. More recently, parametric surfaces have begun to be used in computer-generated special effects for film and art exhibits. Most recently, parametric surfaces have begun to be used in video games. Using parametric surfaces to model objects in video games enhances the experience of playing video games by making objects in the virtual world appear more realistic.

2.1 Applications for NURBS Surfaces

The information in this application note is most applicable to rendering NURBS in real-time applications, such as games. Most video games currently use polygonal models to represent objects in their scenes. NURBS are presented here as an alternative to polygonal models.

Memory Consumption

The amount of memory required to store a NURBS model is constant regardless of the level of detail at which the model is displayed. The object is tessellated on the fly, so the level of detail doesn't affect the amount of memory required to store the object. Polygonal models are fixed, so they require more and more memory with increasing levels of detail.

Level of detail

A NURBS surface definition can be used to approximate the surface it defines to any degree of detail. The same control points and basis functions are used to approximate the surface regardless of the level of detail at which the surface is being displayed. For polygonal models, multiple levels of detail typically require multiple and distinct polygonal meshes. The video game artist must create one instance of the model object for each level of detail that is to be displayed. Many levels of detail can be derived from single NURBS model, so an artist is required to draw a NURBS model only once. This also improves memory performance because switching models is not needed as levels of detail change.

Animation

Animating NURBS models is computationally less expensive than animating polygonal models. Moving the control points of the NURBS surfaces animates NURBS models. Except for very low levels of detail, a NURBS version of a model will have fewer control points than vertices in a polygonal version of the model. To animate the NURBS version of the model many fewer points must be moved. Additionally, for NURBS models exactly the *same* points must be moved regardless of the level of detail at which the model is displayed.

Computation/memory trends

Historically, for personal computers, computational bandwidth has increased much quicker than memory bandwidth. Computational bandwidth refers to the number of instructions executed by a microprocessor per second. Memory bandwidth refers to the number of bytes transferred from the memory subsystem to the microprocessor per second. For the foreseeable future, computational bandwidth will continue to increase faster than memory bandwidth. To fully exploit the resources of future PCs, programs must be designed with this trend in mind.

Polygonal models typically consume more memory resources (they are larger) than NURBS models, but NURBS models consume more computational resources during rendering (they are more expensive to render). At the levels of detail exhibited by objects in video games currently on the market, polygonal models can be rendered faster than NURBS models. However, due to caching effects (large polygonal models do not fit into the cache) and a well-known optimization technique (transforming control points of NURBS surface before tessellating the surface), the speed of rendering NURBS models at high levels of detail approaches and even surpasses the speed at which polygonal models can be rendered. Due to

the trend of computational bandwidth increasing faster than memory bandwidth, the rendering of NURBS models is becoming more efficient and less expensive than the rendering of polygonal models. Eventually, even at the levels of detail seen today, rendering NURBS models can be expected to become less expensive than rendering polygonal models.

Transformation

As discussed in the following section, NURBS surfaces are rational B-splines. Rational curves and surfaces are invariant under translation, scaling, rotation and perspective transformation of their control points. Non-rational curves and surfaces vary under perspective transformations of their control points. This means that the translation, scaling, rotation and perspective transformations may be applied to the control points of a NURBS surface before the surface is tessellated. Transformed surface points are generated when the surface is tessellated using transformed control points. This may translate into a significant performance gain when many surface points are tessellated compared to the number of transformed control points.

2.2 Background of NURBS Surfaces

NURBS surfaces provide more smoothness and control than most other types of parametric surfaces. Cubic surfaces were selected for this study because they typically provide enough smoothness and control of the surface's shape for most applications. Cubic surfaces also fit nicely within the four-wide SIMD implementation of the Intel Pentium III processor.

The degree of the basis functions of a surface determine the degree of the surface. A cubic surface has basis functions of degree 3. Cubic basis functions take the general form:

$$b(u) = a_3*u^3 + a_2*u^2 + a_1*u + a0$$

where:

b(u) is the basis function

u is the parameter on which the basis function is defined

a_3, a_2, a_1, a_0 are the basis coefficients

For more information on basis functions, refer to the appendix of this application note, entitled *Knot Vectors and Basis Functions*.

This section assumes the reader is familiar with the terminology related to curved surfaces. The reader is encouraged to read the chapter on parametric curves and surfaces in *Computer Graphics: Principles and Practice, 2nd Edition* before continuing.

NURBS surfaces are a generalization of nonrational B-splines and rational and nonrational Bezier curves and surfaces. NURBS surfaces have all of the properties of the aforementioned surfaces while overcoming some of their limitations. The most important limitation that NURBS overcomes is control over the continuity of a surface. Nonuniform B-splines can be made to interpolate starting and ending points and control points without introducing linear segments into the spline, whereas uniform B-splines cannot.

There are two significant differences between lighting NURBS surfaces and typical 3D geometry. In order to light transformed surface points, lights must first be transformed using the same transformation that was applied to the control points. This is a relatively inexpensive operation, but it is typically not done this way in 3D geometry pipelines.

The second difference lies in generating surface normals. Surface normals are most commonly used during lighting to determine the orientation of the surface in relation to lights. Like surface points, surface normals must be calculated for curved surfaces. The equation used to calculate surface normals is very similar to the equation used to calculate surface points. The difference is that for surface normals basis function derivatives are used instead of just the basis functions. This subject is outside the scope of this paper. Refer to the references for more information on this topic.

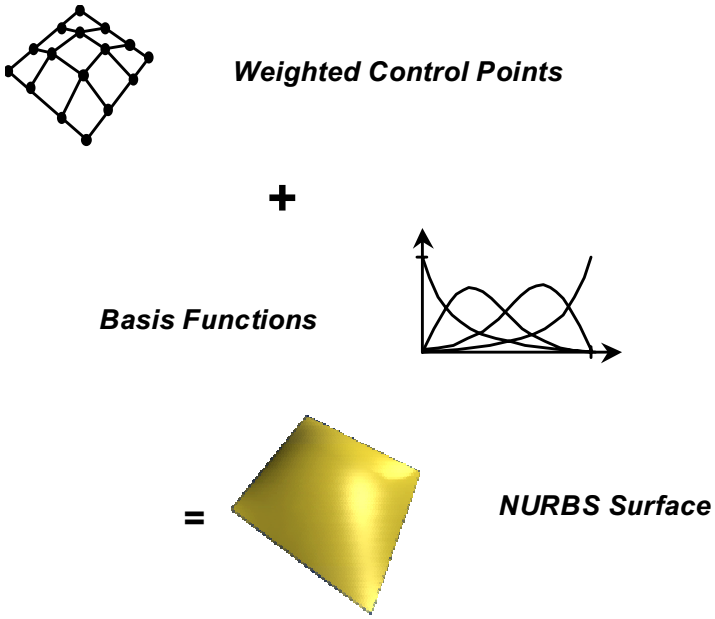

Figure 1: Control points and basis functions define NURBS surfaces

Points on a parametric surface are calculated using a set of basis functions and a set of control points, which are blended to form surface points, as illustrated in Figure 1. A bi-parametric function is a function with two parameters. In this case the parameters are *u* and *v*. The bi-parametric function used to calculate NURBS surface points is:

$$\vec{S}(u,v) = \frac{\sum_{i=0}^{p}\sum_{j=0}^{q} b_{i,p}(u)b_{j,q}(v)\overrightarrow{P_{i,j}}}{\sum_{i=0}^{p}\sum_{j=0}^{q} b_{i,p}(u)b_{j,q}(v)w_{i,j}}$$

where:

u and *v* are the two parameters on which the surface is defined

p is the degree of the basis functions defined on parameter *u* – 3 for cubic

q is the degree of the basis functions defined on parameter *v* – 3 for cubic

$\overrightarrow{P_{i,j}} = [xw_{i,j}\ \ yw_{i,j}\ \ zw_{i,j}]$ are the weighted control points ($w_{i,j}$ is the weight)

$w_{i,j}$ is the weight of control point *i,j*

$b_{i,p}(u)$ are the basis functions for parameter u

$b_{i,q}(v)$ are the basis functions for parameter v

$\vec{S}(u,v) = [x(u,v)\ \ y(u,v)\ \ z(u,v)]$ are the tessellated surface points

Note that the user must set all the components of $\vec{S}(u,v)$.

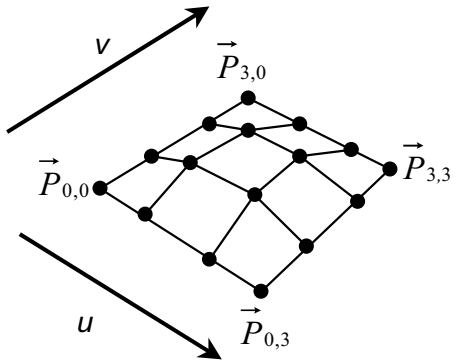

Figure 2: Control Point Grid and the Parameters, u and v

Figure 2 illustrates the relationship between control points and the two parameters, *u* and *v*. Each parameter may have a different number of control points associated with it. The grid in Figure 2 shows a 4x4 grid. Control grids of cubic surface must have at least four control points for each parameter, *u* and *v*. For example, the grid may have any of the following dimensions: 5x4, 10x6, 8x13, etc.

Each parameter, *u* and v, may have a different number of control points and a different degree (*p* may not equal *q*). Each parameter also has its own set of basis functions.

Each dimension of the control point grid is related to two items that define the surface. The first item is the degree of the basis functions. The second item is the number of knot values in the knot vector associated with a given parameter. A knot vector is a non-descending sequence of numbers. The basis functions are derived from knot vectors, so the knot vectors help define the shape of the surface. Knot vectors are discussed in detail in the appendix.

The relationship between the degree of the surface, the number of knot values and the number of control points for parameter *u* is as follows:

$n_u = m_u - p$ where $(m_u + 1) \geq 2 * (p + 1)$

where:

p is the degree of the surface in the u direction

n_u is the number of control points in the u direction

(m_u+1) is the number of knot values in the knot vector associated with parameter u

Similarly, the relationship for parameter *v* is as follows:

$n_v = m_v - q$ where $(m_v + 1) \geq 2 * (q + 1)$

where:

q is the degree of the surface in the v direction

n_v is the number of control points in the v direction

(m_v+1) is the number of knot values in the knot vector associated with parameter v

The important points to remember here are the following:

- u and v have their own knot vectors.
- u and v may have a different number of control points.
- The degree of the surface and the number of knots in the knot vector are related to the number of control points associated with each parameter.

Tessellation

The process of breaking a NURBS surface into a set of discrete polygons is referred to as *tessellation*. After a surface has been tessellated the polygons can be rendered with a standard 3D-geometry pipeline.

As applied in this application note, uniform tessellation does not take the local curvature of a surface into account during tessellation. During uniform tessellation a regular grid of points in parameter space generates a (possibly non-regular) grid of surface points is that approximate the surface. The user (be it human or a program) specifies a step count and range for both the u and the v parameter before the surface is tessellated. The reciprocals of the step counts are parametric step sizes for each parameter. Typically, the parametric range is set to [0,1]. A step count of 10 results in a parametric step size of 1/10 or 0.1. This means that the basis functions will be evaluated at {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}. If the step count and range for both the u and v parameters are set to 10 and [0,1], the following are the uv coordinates at which surface points, $\vec{S}(u,v)$, will be evaluated:

{ (0.0,0.0), (0.0,0.1), (0.0,0.2), …, (0.0,0.8), (0.0,0.9), (0.0,1.0),
(0.1,0.0), (0.1,0.1), (0.1,0.2), …, (0.1,0.8), (0.1,0.9), (0.1,1.0),
(0.2,0.0), (0.2,0.1), (0.2,0.2), …, (0.2,0.8), (0.2,0.9), (0.2,1.0),
.
.
.
(0.8,0.0), (0.8,0.1), (0.8,0.2), …, (0.8,0.8), (0.8,0.9), (0.8,1.0),
(0.9,0.0), (0.9,0.1), (0.9,0.2), …, (0.9,0.8), (0.9,0.9), (0.9,1.0),
(1.0,0.0), (1.0,0.1), (1.0,0.2), …, (1.0,0.8), (1.0,0.9), (1.0,1.0) }

For this application note uniform tessellation was selected over non-uniform tessellation because of the relative simplicity of the algorithm and because uniform tessellation is more suited for SIMD processing. Non-uniform tessellation can generate more efficient polygonal representations of NURBS surfaces, but the algorithm is riddled with unpredictable branches and, depending on how it is implemented, recursive calculations. Keep in mind that for real world applications, a combination of uniform and non-uniform tessellation may be required to achieve the desired level of detail without additional overhead either on bandwidth or computation.

The following is pseudo-code for uniform NURBS surface tessellation:

```
// algorithm for cubic NURBS surface tessellation.
float u = v = 0.0;                  // parametric values
```

```
float u_step_size = 1.0 / u_steps;  // u direction parametric step
float v_step_size = 1.0 / v_steps;  // v direction parametric step

// u_steps is the number of parametric steps to take in the u direction
for ( int i = 0; i <= u_steps; i++ )
{
  // v_steps is the number of parametric steps to take in the v direction
  // (u_steps*v_steps) is the number of surface points to evaluate
  for ( int j = 0; j <= v_steps; j++ )
  {
```

$$\vec{S}(u,v) = \frac{\sum_{i=0}^{p}\sum_{j=0}^{q} b_{i,p}(u)\,b_{j,q}(v)\,\overrightarrow{P_{i,j}}}{\sum_{i=0}^{p}\sum_{j=0}^{q} b_{i,p}(u)\,b_{j,q}(v)\,w_{i,j}}$$

```
    u += u_step_size;
  }
  v += v_step_size;
}
```

2.3 Implementing Cubic Surface Tessellation

This section provides guidelines for implementing cubic NURBS uniform surface subdivision with and without SSE optimizations. Be mindful at all times that the only difference between the implementations presented here lies in how $\vec{S}(u,v)$ is evaluated.

Each implementation discussed in this application note fills an array in memory with surface points. A sample data structure for surface points follows:

```
struct SurfacePoint{
  float x, y, z;
};
```

The implementation that uses the SSE instructions requires the following control point data structure:

```
struct ControlPoint {
  float x, y, z, w;
};
```

The other implementations do not require this data structure, but use it anyway.

Knot vectors and knot spans are discussed in great detail in the appendix. All of the implementations use an array of knot span indices to determine which set of control points to use during surface evaluation; a set of (p+1)x(q+1) control points are used to evaluate any one surface point. The array of knot span indices is generated before surface tessellation takes place.

2.3.1 Improving the Equation

A more efficient version of $\vec{S}(u,v)$ (shown previously) is used on all three implementations. The equation for a surface point, $\vec{S}(u,v)$, in homogeneous coordinates follows:

$$\overrightarrow{S_w}(u,v) = \sum_{i=0}^{p}\sum_{j=0}^{q} b_{i,p}(u)b_{j,q}(v)\overrightarrow{P_{i,j}}$$

$\overrightarrow{S_w}(u,v)$ is the surface point in homogeneous coordinates and has the components [*X Y Z W*].

The non-homogeneous surface point, $\vec{S}(u,v)$, is calculated from the homogeneous surface point, $\overrightarrow{S_w}(u,v)$, using the following equation:

$$\vec{S}(u,v) = \begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} \frac{X}{W} & \frac{Y}{W} & \frac{Z}{W} \end{bmatrix}$$

X, Y, Z and W are the components of $\overrightarrow{S_w}(u,v)$.

This set of equations for evaluating surface points is suitable for use in both the Intel® x87 architecture implementations and the implementations with SSE instructions.

The cubic version of the equation has the degree fixed to three. The cubic equation is as follows:

$$\overrightarrow{S_w}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_{i,3}(u)b_{j,3}(v)\overrightarrow{P_{i,j}}$$

$$\vec{S}(u,v) = \begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} \frac{S_{W,X}(u,v)}{S_{W,W}(u,v)} & \frac{S_{W,Y}(u,v)}{S_{W,W}(u,v)} & \frac{S_{W,Z}(u,v)}{S_{W,W}(u,v)} \end{bmatrix}$$

This equation is an improvement because it is more suited to evaluation using the SSE instructions than the original version of the equation. If all four coordinates (x, y, z, w) are calculated simultaneously, there are always four independent mathematical operations that can be performed using the SSE instructions.

2.3.2 Standard Intel® x87 Architecture Implementations

Three Intel x87 architecture implementations of uniform NURBS surface tessellation are presented in this application note. The first implementation may be used to tessellate NURBS surfaces of any degree. This implementation is a direct application of the equations discussed in the previous section. The summations in $\overrightarrow{S_w}(u,v)$ are implemented as a pair of nested loops.

The first cubic implementation is very similar to the previous implementation. The difference is that the variables, which indicate the degree of the surface being tessellated, are replaced by 3.

The difference between the two cubic versions is in the implementation of the summations in $\overrightarrow{S_w}(u,v)$. The first cubic version implements the summations as a pair of nested loops. The second cubic version unrolls those inner loops.

Source code for each of these versions is found in the appendices.

2.3.3 Implementations with SSE

The implementations that use SSE instructions are derived from the Intel x87 architecture implementations. There are two SSE implementations. The first one is derived from the Intel x87 architecture implementation that evaluates any degree surface. The second implementation is derived from the cubic Intel x87 architecture implementation with unrolled inner loops.

Both SSE implementations simultaneously calculate the four components of the homogeneous surface point, or $\overrightarrow{S_w}(u,v)$. The equations for each of the components are:

$$X = \sum_{i=0}^{3}\sum_{j=0}^{3} b_{i,3}(u)b_{j,3}(v)x_{i,j}$$

$$Y = \sum_{i=0}^{3}\sum_{j=0}^{3} b_{i,3}(u)b_{j,3}(v)y_{i,j}$$

$$Z = \sum_{i=0}^{3}\sum_{j=0}^{3} b_{i,3}(u)b_{j,3}(v)z_{i,j}$$

$$W = \sum_{i=0}^{3}\sum_{j=0}^{3} b_{i,3}(u)b_{j,3}(v)w_{i,j}$$

where

X, Y, Z, W are the components of the surface point in homogeneous space, or $\overrightarrow{S_w}(u,v)$

All four of these equations are evaluated simultaneously in XMM registers. At the end of the evaluation each element of an XMM register contains one of the components of $\overrightarrow{S_w}(u,v)$. One element contains the value for X, one element contains the value for Y, and likewise for Z and W.

The homogeneous surface point (xw, yw, zw as mentioned above) is turned into a non-homogeneous surface point by multiplying the components X, Y and Z by $1/W$. This is an opportunity for optimization using the SSE instructions. First, a copy is made of the register that contains X, Y, Z and W. Then W is broadcasted across the copy so that each element contains the value of W. The reciprocal approximation instruction (*rcpps*) is used to calculate $1/W$ in each element of this register. Finally, the contents of this register are multiplied by the contents of the register that contains X, Y, Z and W. The result is the surface point in non-homogeneous space. The following pseudo-code illustrates this procedure using SSE intrinsics:

```
// pseudo-code for calculating non-homogeneous surface point from
// the homogeneous surface point
__m128 XYZW, rcpW, xyz1;                 // XYZW = [ W    Z    Y    X ]
rcpW = _mm_shuffle_ps(XYZW, XYZW, 0xff); // rcpW = [ W    W    W    W ]
rcpW = _mm_rcp_ps(rcpW);                 // rcpW = [1/W  1/W  1/W  1/W]
xyz1 = _mm_mul_ps(rcpW, XYZW);           // xyz1 = [ 1    z    y    x ]
```

After x, y and z have been calculated they must be stored to an array of surface points. To store x and y, you can use the *_mm_storel_pi*() intrinsic. To store z, use the *_mm_movehl_ps*() intrinsic to move the z value to the least significant element of the register (the scalar element) and then a do a scalar store using *_mm_store_ss*(). Here is the pseudo-code for this step:

```
// pseudo-code for storing non-homogeneous surface point to
```

```
// surface point array
SurfacePoint *S; // pointer to current point in surface point array
                 // struct SurfacePoint { x, y, z };
_mm_storel_ps((__m64*)S, xyz1);
_mm_store_ss(S->z, _mm_movehl_ps(xyz1, xyz1));
```

In the any-degree surface implementation the summations in $\overrightarrow{S_w}(u,v)$ are implemented as two nested loops. In the implementation that evaluates cubic surfaces, those inner loops are unrolled. This is the primary difference between the two SSE implementations, which can be seen in the source in the appendices.

2.3.4 Tips and Tricks

Reciprocal Approximation

A trade of accuracy for performance is made in the calculation of reciprocal homogeneous W or 1/W. The divide approximation SSE instruction, *rcpps*, is used to calculate 1/W instead of the SSE divide instruction, *divps*. The divide approximation instruction approximates a 1/x operation. Using *rcpps* instead of *divps* results in a significant performance improvement in the divide. The instruction *divps* has a very long latency and throughput compared to *rcpps*. The error of *rcpps* is $1.5 * 2^{-12}$. If more accuracy is needed, use the Newton-Raphson algorithm instead of *divps*. See the references section for more information about the Newton-Raphson algorithm.

Data Alignment

Aligned loads are performed by using the *movaps* instruction. Unaligned loads are performed by the *movups* instruction. Aligned loads are faster than unaligned loads. Aligning input data to 16 byte boundaries enables the use of aligned loads. For better performance, the control point array and the basis function array should be aligned. The size of the data structure for control points is 16 bytes (4 floats at 4 bytes/float). A data structure that is 16 bytes long is ideal for the SSE implementation. If the first element in an array of these data structures is aligned to a 16 byte boundary then all of the elements of that array will be aligned to 16 byte boundaries. This enables aligned memory accesses to be performed on the data in the array. The hardware has special support for aligned memory accesses that increases the performance of aligned memory accesses over unaligned memory accesses. Intrinsics such as *_mm_load_ps()* and *_mm_store_ps()* exploit the hardware support of aligned memory accesses.

Basis values are the values obtained by evaluating the basis functions. They are used to evaluate $\vec{S}(u,v)$. By aligning the basis value arrays, aligned loads can be used on the basis values. For each parametric value at which the surface is to be evaluated in the *u* and *v* directions there are four basis values. Each basis value is a `float`; thus, a 16-byte long data structure can be used to hold each set of four basis values. Similar to the control point array, aligning the first elements in the basis value arrays results in all of the elements in the arrays being aligned; thus, aligned memory accesses may be used on the elements of the basis value arrays.

Calculating all four components (X, Y, Z and W) of the homogeneous surface point simultaneously enables full use of all four components of the XMM registers. Evaluating $\overrightarrow{S_w}(u,v)$ before moving the surface point to non-homogeneous coordinates requires that four coordinates be calculated instead of three. This enables all four elements of every XMM register to be fully used during surface point calculation.

3 Performance

Obviously, surface point evaluation is not free, so all efforts must be taken to ensure that the minimum number of surface points are evaluated at all times. The minimum number of surface points is the number of surface points required to represent the surface at a given level of detail. The user or the developer determines this number visually.

The amount of time required to evaluate one surface point was used as the performance indicator during this study. This number was calculated by first measuring the total elapsed time during surface evaluation (time required to evaluate all of the surface points). Then, the average time required to evaluate one surface point was calculated by dividing the total elapsed time by the number of surface points evaluated. Performance comparisons were done on a per-surface-point basis.

Surface characteristics and the degree of tessellation cause performance variations. Control points must be added to surface definitions to add surface details. The more detail expressed by a surface, the more control points required in the surface definition. The number of control points in the surface definition impacts performance of surface tessellation. The more control points in the definition, the poorer the performance of surface tessellation. The performance decreases because the memory required to store the surface definition increases and the logic required to determine the correct set of control points to use in surface point evaluation is more expensive. Artists should be made aware of this relationship. Artists can use control points wisely by using the minimum *required* number of control points to define the object they are creating.

The degree of tessellation also causes performance variations. Generally, per-surface point performance improves as the number of surface points evaluated increases. There is more than one reason for this. First, the same amount of data must be loaded into the cache regardless of the number of surface points to be evaluated. The memory load overhead incurred during evaluating a 100-point surface is approximately the same as that incurred during the evaluation of 1000 points.

Secondly, the accuracy of branch prediction improves as more surface points are evaluated. The number of instructions executed per branch increases as more surface points are evaluated. This means that more time is spent actually executing instructions instead of recovering from mispredicted branches.

3.1 Gains/Improvements

The cubic SSE implementation provided the fastest solution. Its performance is significantly better than the SSE implementation that can tessellate surfaces of any degree. The performances of both SSE implementations are significantly faster than the Intel x87 architecture implementations. Each implementation's performance was improved by roughly the same amount on the Pentium 4 processor compared to the Pentium III processor.

3.2 Considerations

NURBS surface tessellation performance can vary widely depending on how it is implemented. There is always a trade-off between development time and performance for NURBS surface tessellation. Typically, encoding an algorithm in assembly code consumes more time and effort than encoding an algorithm in high level languages, such as C and C++. The SSE intrinsics (provided with the Intel® C/C++ Compiler) were used in all of the implementations that use SSE instructions. The SSE intrinsics

provide a middle ground between assembly code and C. The SSE intrinsics is a C++ API that provides a direct interface to the SSE. The performance gain due to using the intrinsics was roughly equivalent to that of encoding NURBS surface tessellation with assembly code. This was verified by examining the assembly code generated by compiling the intrinsics implementation of NURBS surface tessellation.

Fixing the degree of the NURBS surface to be defined as cubic, as in the cubic Intel x87 architecture and cubic SSE implementations, is a trade-off between generality and performance. Any surface can be defined by cubic NURBS surfaces. If a higher degree of continuity is required, the artist can draw the surface using higher-order surfaces. After the work is completed, a post-processing step may be performed on the surface. During this step, the higher order surface may be split into multiple cubic surfaces that, together, have the same shape as the higher-order surfaces. Details on how to implement this post-processing step are found in *The NURBS Book* by Piegl and Tiller (see the References section).

4 Conclusion

Constraining the degree of a NURBS surface to three increases the performance of surface tessellation by enabling the following:

- removal of all branches from the inner-most loops of surface point tessellation.
- alignment of basis values for the SSE implementations.

The SSE instructions provide several features that enable increased performance of applications that display NURBS surfaces. These are the main reasons for performance gain in applications using the SSE:

- SIMD feature of these instructions.
- *rcpps* instruction can replace the long latency divps instructions, in places where an increase in error (compared to *divps*) is acceptable.
- performing aligned memory accesses on control points and basis function values.
- fewer overall instructions to execute compared to the Intel x87 architecture versions.

The Pentium 4 processor exhibits a significant performance gain compared to the Pentium III processor. The main reason for the performance gain is its far higher microprocessor frequency. All of the implementations of surface tessellation discussed in this document are scaled almost equally with processor frequency.

5 Code Examples

The code samples that follow require the following macros to be defined:

```
#define CUBIC_DEGREE 3
#define CUBIC_ORDER 4

The following code also requires the following structures to be defined:

struct SurfacePoint {
    float x, y, z;
};
```

5.1 Standard Intel x87 Architecture Coding Examples

The following C coding examples standard Intel x87 architecture floating point operations to implement uniform tessellation of NURBS surfaces.

5.1.1 Tessellation of NURBS Surfaces of Any Degree

The following C code performs uniform NURBS surface tessellation on surfaces of any degree.

```
 void TessellateSurfaceC(
    SurfacePoint surface_points,    // output surface points
    unsigned int u_step_count,      // steps to take in u direction
    unsigned int v_step_count,      // steps to take in v direction
    unsigned int u_degree,          // degree of basis values in u direction
    unsigned int v_degree,          // degree of basis values in v direction
    float *u_basis_values,          // values from u basis functions
    float *v_basis_values,          // values from v basis functions
    unsigned int u_knot_spans,      // knot span indices for u values
    unsigned int v_knot_spans)      // knot span indices for v values
 {
     SurfacePoint *sp = surface_points;
     float *u_basis = u_basis_values;
     unsigned int *u_knot_span = u_knot_span_indices;

     /* for each parametric value in the u direction */
     for ( unsigned int u_index = 0; u_index < u_step_count; u_index++ )
     {
         float *v_basis = v_basis_values;
         unsigned int *v_knot_span = v_knot_spans;
```

```
/* for each parametric value in the v direction */
for ( unsigned int v_index = 0; v_index < v_step_count; v_index++ )
{
    /* this is a surface point in homogeneous space */
    /* it holds intermediate values until the end of the function */
    struct HomogeneousSurfacePoint
    {
        float X, Y, Z, W;
    };

    HomogeneousSurfacePoint hsp;
    hsp.X = hsp.Y = hsp.Z = hsp.W = 0.0f;

    /*
     Step through the control points row-by-row, column-by-
     column. Here we calculate where the beginning of the first row
     for the outer loop.  This pointer (cp_row) is incremented one
     row at a time.
     */
    ControlPoint *cp_row = &control_points[
        ((*u_knot_span)-u_degree) * v_control_point_count +
        ((*v_knot_span)-v_degree)];

    /* calculate the surface point in homogeneous space */
    for ( unsigned int i = 0; i <= u_degree; i++ )
    {
        /* current control point is the first in the current row */
        ControlPoint *cp = cp_row;

        /* reset temp to zero */
        HomogeneousSurfacePoint temp;
        temp.X = temp.Y = temp.Z = temp.W = 0.0f;

        /* inner summation */
        for ( unsigned int j = 0; j <= v_degree; j++ )
        {
            temp.X += v_basis[j] * cp->x;
            temp.Y += v_basis[j] * cp->y;
            temp.Z += v_basis[j] * cp->z;
            temp.W += v_basis[j] * cp->w;
```

```
                    /* increment to the next control point in the row */
                    cp++;
                }

                /* outer sum */
                hsp.X += u_basis[i] * temp.X;
                hsp.Y += u_basis[i] * temp.Y;
                hsp.Z += u_basis[i] * temp.Z;
                hsp.W += u_basis[i] * temp.W;

                /* increment to the next row of control points */
                cp_row += v_control_point_count;
            }

            /* bring homogeneous point into nonhomogeneous space */
             hsp.W = 1.0f / hsp.W;
            sp->x = hsp.X * hsp.W;
            sp->y = hsp.Y * hsp.W;
            sp->z = hsp.Z * hsp.W;

             /* increment to next surface point in surface point array */
            sp++;

            /* increment to next set of v basis values */
            v_basis += (v_degree+1);

             /* increment to the next v knot span index */
            v_knot_span++;
        }

         /* increment to next set of u basis values */
        u_basis += (u_degree+1);

        /* increment to the next u knot span index */
        u_knot_span++;
    }
}
```

5.1.2 Tessellation of Cubic NURBS Surfaces with Nested Inner Loops

The following C code performs uniform NURBS surface tessellation on cubic surfaces.

```
void TessellateCubicSurfaceC(
    SurfacePoint surface_points,   // output surface points
    unsigned int u_step_count,     // steps to take in u direction
    unsigned int v_step_count,     // steps to take in v direction
    unsigned int u_degree,         // degree of basis values in u direction
    unsigned int v_degree,         // degree of basis values in v direction
    float *u_basis_values,         // values from u basis functions
    float *v_basis_values,         // values from v basis functions
    unsigned int u_knot_spans,     // knot spans for u values
    unsigned int v_knot_spans)     // knot spans for v values
 {
    if ( u_degree != CUBIC_DEGREE || v_degree != CUBIC_DEGREE )
    {
      exit 0;
    }

    SurfacePoint *sp = surface_points;    // current output surface point
    float *u_basis = u_basis_values;      // current set of 4 u basis values
    unsigned int *u_knot_span = u_knot_spans; // current u knot span index

     /* for each parametric value in the u direction */
     for ( unsigned int u_index = 0; u_index < u_step_count; u_index++ )
     {
         float *v_basis = v_basis_values;       // current v basis values
         unsigned int *v_knot_span = v_knot_spans; // current v knot span idx

         /* for each parametric value in the v direction */
         for ( unsigned int v_index = 0; v_index < v_step_count; v_index++ )
         {
             /* this is a surface point in homogeneous space */
             /* it holds intermediate values until the end of the function */
             struct HomogeneousSurfacePoint
             {
                 float X, Y, Z, W;
             };
```

```
HomogeneousSurfacePoint hsp;
hsp.X = hsp.Y = hsp.Z = hsp.W = 0.0f;

/* step through control points row-by-row, column-by-column */
/* here we calculate where the beginning of the first row is */
/* in the outer summation loop this pointer is incremented by
   one row of control points at a time */
ControlPoint *cp_row = &control_points[
    ((*u_knot_span)-CUBIC_DEGREE) * v_control_point_count +
    ((*v_knot_span)-CUBIC_DEGREE)];

/* calculate the surface point in homogeneous space */
for ( unsigned int i = 0; i <= CUBIC_DEGREE; i++ )
{
    /* current control point is the first in the current row */
    ControlPoint *cp = cp_row;

    /* reset temp to zero */
    HomogeneousSurfacePoint temp;
    temp.X = temp.Y = temp.Z = temp.W = 0.0f;

    /* inner summation */
    for ( unsigned int j = 0; j <= CUBIC_DEGREE; j++ )
    {
        temp.X += v_basis[j] * cp->x;
        temp.Y += v_basis[j] * cp->y;
        temp.Z += v_basis[j] * cp->z;
        temp.W += v_basis[j] * cp->w;

        /* increment to the next control point in the row */
        cp++;
    }

    /* outer summation */
    hsp.X += u_basis[i] * temp.X;
    hsp.Y += u_basis[i] * temp.Y;
    hsp.Z += u_basis[i] * temp.Z;
```

```
                hsp.W += u_basis[i] * temp.W;

                /* increment to the next row of control points */
                cp_row += v_control_point_count;
            }

            /* bring control point into non-homogeneous space */
            hsp.W = 1.0f / hsp.W;  // calculate reciprocal W
            sp->x = hsp.X * hsp.W;
            sp->y = hsp.Y * hsp.W;
            sp->z = hsp.Z * hsp.W;

            /* increment to the next output surface point */
            sp++;

            /* increment to the next set of v basis values */
            v_basis += CUBIC_ORDER;

            /* increment to the next v knot span index */
            v_knot_span++;
        }

        /* increment to the next set of u basis values */
        u_basis += CUBIC_ORDER;

        /* increment to the next u knot span index */
        u_knot_span++;
    }
}
```

5.1.3 Tessellation of Cubic NURBS Surfaces with Unrolled Inner Loops

The following C code performs uniform NURBS surface tessellation on cubic surfaces. The summations on the innermost loops are unrolled.

```
void TessellateCubicSurfaceC_Unrolled(
    SurfacePoint surface_points,    // output surface points
    unsigned int u_step_count,      // steps to take in u direction
    unsigned int v_step_count,      // steps to take in v direction
    unsigned int u_degree,          // degree of basis values in u direction
    unsigned int v_degree,          // degree of basis values in v direction
    float *u_basis_values,          // values from u basis functions
    float *v_basis_values,          // values from v basis functions
    unsigned int u_knot_spans,      // knot spans for u values
    unsigned int v_knot_spans)      // knot spans for v values
 {
     if ( u_degree != CUBIC_DEGREE || v_degree != CUBIC_DEGREE )
     {
         exit 0;
     }

     SurfacePoint *sp = surface_points; // current output surface point
     float *bu = u_basis_values;// current set of u basis values
     unsigned int *u_knot_span = u_knot_spans; // current u knot span index

     /* for each parametric value in the u direction */
     for ( unsigned int u_index = 0; u_index < u_step_count; u_index++ )
     {
         float *bv = v_basis_values; // current set of v basis values
         unsigned int *v_knot_span = v_knot_spans; // current v knot span idx

         /* for each parametric value in the v direction */
         for ( unsigned int v_index = 0; v_index < v_step_count; v_index++ )
         {
             /* this is a surface point in homogeneous space */
             /* it holds intermediate values until the end of the function */
             struct HomogeneousSurfacePoint
             {
                 float X, Y, Z, W;
```

```
            };

            HomogeneousSurfacePoint hsp;
            hsp.X = hsp.Y = hsp.Z = hsp.W = 0.0f;

            ControlPoint *cp = &control_points[
                ((*u_knot_span)-CUBIC_DEGREE) * v_control_point_count +
                ((*v_knot_span)-CUBIC_DEGREE)];

#define HOMOGENEOUS_SURFACE_POINT(idx)                                      \
hsp.X += bu[idx]*(bv[0]*cp[0].x+bv[1]*cp[1].x+bv[2]*cp[2].x+bv[3]*cp[3].x);\
hsp.Y += bu[idx]*(bv[0]*cp[0].y+bv[1]*cp[1].y+bv[2]*cp[2].y+bv[3]*cp[3].y);\
hsp.Z += bu[idx]*(bv[0]*cp[0].z+bv[1]*cp[1].z+bv[2]*cp[2].z+bv[3]*cp[3].z);\
hsp.W += bu[idx]*(bv[0]*cp[0].w+bv[1]*cp[1].w+bv[2]*cp[2].w+bv[3]*cp[3].w);\
cp += v_control_point_count;

            HOMOGENEOUS_SURFACE_POINT(0)
            HOMOGENEOUS_SURFACE_POINT(1)
            HOMOGENEOUS_SURFACE_POINT(2)
            HOMOGENEOUS_SURFACE_POINT(3)

            // bring the point into nonhomogeneous space
            hspW = 1.0f / hsp.W;
            sp->x = hsp.X * hsp.W;
            sp->y = hsp.Y * hsp.W;
            sp->z = hsp.Z * hsp.W;

            // increment to the next output surface point
            sp++;

            // increment to the next set of u basis values
            bv += CUBIC_ORDER;

            // increment to the next v knot span index
            v_knot_span++;
        }

        // increment to the next set of v basis values
```

```
            bu += CUBIC_ORDER;

            // increment to the next u knot span index
            u_knot_span++;
        }
    }
```

5.2 SSE Technology Intrinsics Coding Examples

The C code in this section demonstrates how to use SSE instructions in the uniform tessellation of NURBS surfaces.

But first, you must define these macros:

```
/* checks to see if pointer 'p' is aligned to a 16 byte boundary */
#define IS_ALIGNED_16(p) (((DWORD)(p))%16 == 0)

/* the number of coordinates per surface point (x, y, z) */
#define COORD_PER_SURF_PT 3
```

5.2.1 Tessellation of NURBS Surfaces using SSE Intrinsics

SSE intrinsics are used in the following C code to perform uniform tessellation on NURBS surfaces of any degree.

```
void TessellateSurfaceSSE(
    SurfacePoint surface_points,   // output surface points
    unsigned int u_step_count,     // steps to take in u direction
    unsigned int v_step_count,     // steps to take in v direction
    unsigned int u_degree,         // degree of basis values in u direction
    unsigned int v_degree,         // degree of basis values in v direction
    float *u_basis_values,         // values from u basis functions
    float *v_basis_values,         // values from v basis functions
    unsigned int u_knot_spans,     // knot spans for u values
    unsigned int v_knot_spans)     // knot spans for v values
 {
     float *sp = (float*) surface_points; // current output surface point
     float *u_basis = u_basis_values;     // current set of u basis values
     unsigned int *u_knot_span = u_knot_spans; // current u knot span index

     /* for each parametric value in the u direction */
     for ( unsigned int u_index = 0; u_index < u_step_count; u_index++ )
     {
         float *v_basis = v_basis_values; // current set of v basis values
         unsigned int *v_knot_span = v_knot_spans; // current v knot span idx

         /* for each parametric value in the v direction */
```

```
for ( unsigned int v_index = 0; v_index < v_step_count; v_index++ )
{
    /* hsp --> surface point in homogeneous space */
    __m128 hsp = _mm_setzero_ps(); // init all elements to zero

    /* Step through control points row-by-row, column-by-column. */
    /* Calculate where the beginning of the first row is. */
    /* In the outer summation loop this pointer is incremented one
       row at a time */
    ControlPoint *cp_row = &control_points[
        ((*u_knot_span)-u_degree) * v_control_point_count +
        ((*v_knot_span)-v_degree)];

    /* calculate the surface point in 3D homogeneous space */

    for ( unsigned int i = 0; i <= u_degree; i++ )
    {
        /* current control point is first in the current row */
        float *cp = (float*) cp_row;
        ASSERT( IS_ALIGNED_16(cp) );

        __m128 temp = _mm_setzero_ps(); // intermediate values
        __m128 basis;   // temporarily holds basis values

        /* inner summation */
        for ( unsigned int j = 0; j <= v_degree; j++ )
        {
            temp = _mm_add_ps(temp,
                    _mm_mul_ps(
                        _mm_set_ps1(v_basis[j]),
                        _mm_load_ps(cp)));

            /* increment to the next control point in the row */
            cp += 4;
        }

        /* outer summation */
        hsp = _mm_add_ps(hsp,
```

```
                    _mm_mul_ps(_mm_set_ps1(u_basis[i]), temp));

                /* increment to the next row of control points */
                cp_row += v_control_point_count;
            }

            /* calculate reciprocol homogeneous W  -->  1/W */
            __m128 rhw = _mm_rcp_ps(_mm_shuffle_ps(hsp, hsp, 0xff));

            /* bring the surface point into nonhomogeneous space */
            hsp = _mm_mul_ps(hsp, rhw);

            /* store the surface point to the surface point array */
            _mm_storel_pi((__m64*) sp, hsp);
            _mm_store_ss(sp+2, _mm_movehl_ps(hsp, hsp));

            /* increment to next output surface point */
            sp += COORD_PER_SURF_PT;

            /* increment to next set of v basis values */
            v_basis += (v_degree+1);

            /* increment to next v knot span index */
            v_knot_span++;
        }

        /* increment to next set of u basis values */
        u_basis += (u_degree+1);

        /* increment to next u knot span index */
        u_knot_span++;
    }
}
```

5.2.2 Tessellation of Cubic NURBS Surfaces Using SSE Intrinsics

The SSE intrinsics are used in the following C code to perform uniform tessellation on cubic NURBS surfaces.

```
void TessellateCubicSurfaceSSE(
    SurfacePoint surface_points,   // output surface points
    unsigned int u_step_count,     // steps to take in u direction
    unsigned int v_step_count,     // steps to take in v direction
    unsigned int u_degree,         // degree of basis values in u direction
    unsigned int v_degree,         // degree of basis values in v direction
    float *u_basis_values,         // values from u basis functions
    float *v_basis_values,         // values from v basis functions
    unsigned int u_knot_spans,     // knot spans for u values
    unsigned int v_knot_spans)     // knot spans for v values
 {
     float *sp = (float*) surface_points; // current output surface point
     float *u_basis = u_basis_values; // current set of u basis values
     unsigned int *u_knot_span = u_knot_spans; // current u knot span index

     /* for each parametric value in the u direction */
     for ( unsigned int u_index = 0; u_index < u_step_count; u_index++ )
     {
         float *v_basis = v_basis_values; // current v basis values
         unsigned int *v_knot_span = v_knot_spans; // current v knot span idx

         /* load and broadcast the basis values */
         ASSERT( IS_ALIGNED_16(u_basis) );
         __m128 bu[CUBIC_ORDER];
         bu[3] = _mm_load_ps(u_basis);
         bu[0] = _mm_shuffle_ps(bu[3], bu[3], 0x00);
         bu[1] = _mm_shuffle_ps(bu[3], bu[3], 0x55);
         bu[2] = _mm_shuffle_ps(bu[3], bu[3], 0xaa);
         bu[3] = _mm_shuffle_ps(bu[3], bu[3], 0xff);

         /* for each parametric value in the v direction */
         for ( unsigned int v_index = 0; v_index < v_step_count; v_index++ )
         {
             /* load and broadcast the basis values */
```

```
            ASSERT( IS_ALIGNED_16(v_basis) );
            __m128 bv[CUBIC_ORDER];
            bv[3] = _mm_load_ps(v_basis);
            bv[0] = _mm_shuffle_ps(bv[3], bv[3], 0x00);
            bv[1] = _mm_shuffle_ps(bv[3], bv[3], 0x55);
            bv[2] = _mm_shuffle_ps(bv[3], bv[3], 0xaa);
            bv[3] = _mm_shuffle_ps(bv[3], bv[3], 0xff);

            /* get a pointer to the first row of control points */
            /* we process the control points one row at a time */
            ControlPoint *cp_row = &control_points[
                ((*u_knot_span)-CUBIC_DEGREE) * v_control_point_count +
                ((*v_knot_span)-CUBIC_DEGREE)];

            /* calculate the surface point in 3D homogeneous space */
            __m128 hsp = _mm_setzero_ps();
            float *cp;

#define HOMOGENEOUS_SURFACE_POINT(idx)                                  \
                                                                        \
    cp = (float*) &cp_row[idx*v_control_point_count];                   \
    ASSERT( IS_ALIGNED_16(cp) );                                        \
                                                                        \
    hsp = _mm_add_ps(hsp,                                               \
            _mm_mul_ps(bu[(idx)],                                       \
                _mm_add_ps(                                             \
                    _mm_add_ps(                                         \
                        _mm_mul_ps(bv[0], _mm_load_ps(cp+0)),           \
                        _mm_mul_ps(bv[1], _mm_load_ps(cp+4))),          \
                    _mm_add_ps(                                         \
                        _mm_mul_ps(bv[2], _mm_load_ps(cp+8)),           \
                        _mm_mul_ps(bv[3], _mm_load_ps(cp+12))))));

            HOMOGENEOUS_SURFACE_POINT(0)
            HOMOGENEOUS_SURFACE_POINT(1)
            HOMOGENEOUS_SURFACE_POINT(2)
            HOMOGENEOUS_SURFACE_POINT(3)
```

```
            /* calculate reciprocol homogeneous W  -->  1/W */
            __m128 rhw = _mm_rcp_ps(_mm_shuffle_ps(hsp, hsp, 0xff));

            /* bring the surface point into nonhomogeneous space */
            hsp = _mm_mul_ps(hsp, rhw);

            /* store the surface point to the surface point array */
            _mm_storel_pi((__m64*) sp, hsp);
            _mm_store_ss(sp+2, _mm_movehl_ps(hsp, hsp));

            /* increment to the next output surface point */
            sp += COORD_PER_SURF_PT;

            /* increment to the next set of v basis values */
            v_basis += CUBIC_ORDER;

            /* increment to the next v knot span index */
            v_knot_span++;
        }

        /* increment to the next set of v basis functions */
        u_basis += CUBIC_ORDER;

        /* increment to the next u knot span index */
        u_knot_span++;
    }
}
```

Appendix A - Knot Vectors and Basis Functions

The basis functions ($b_{i,p}(u)$ and $b_{j,q}(u)$) are derived from knot vectors. Each parameter, u and v, has its own knot vector; therefore, each parameter has its own set of basis functions. The algorithm used to derive the basis functions from the knot vector is the same for both parameters. From here on, unless otherwise stated, references will be made exclusively to parameter u. This is done in an attempt to simplify the discussion – the same procedures and equations apply similarly to parameter v.

The degree of a surface refers to the degree of the basis functions used to define the surface. Basis functions are polynomials and take the following form:

$$b_{i,p}(u) = a_{i,p} * u^{p} + a_{i,p-1} * u^{p-1} + a_{i,p-2} * u^{p-2} + ... + a_{i,2} * u^{2} + a_{i,1} * u + a_{i,0}$$

A cubic surface is defined by third degree basis functions ($p = 3$), which take the following form:

$$b_{i,3}(u) = a_{i,3} * u^{3} + a_{i,2} * u^{2} + a_{i,1} * u + a_{i,0}$$

The coefficients $a_{i,p}$, $a_{i,p-1}$, $a_{i,p-2}$, …, $a_{i,2}$, $a_{i,1}$, $a_{i,0}$ are referred to as *basis coefficients* and are derived from a non-descending sequence of numbers called a *knot vector*. Each number in the sequence that makes up the knot vector is referred to as a *knot* or *knot value*. Knot vectors take the following form:

$U = [u_0, u_1, u_2, ..., u_{m-2}, u_{m-1}]$

where m is the number of knots in the knot vector and is calculated by the following equation:

$m = n + p + 1$

where n is the number of control points, p is the degree of the surface and u_0, u_1, u_2, …, u_{m-2}, u_{m-1} are knot values in the range [0,1].

Each basis function influences the shape of the surface (over a given range of the parameter on which it is defined). A knot determines the parametric value at which one basis function turns off and the next basis function turns on. Restated, the point at which the influence of one basis function stops and the influence of another basis function begins is demarcated by knot values in the knot vector.

There is no single set of basis functions for NURBS as there are for other types of parametric surfaces. Instead, the basis functions depend on the intervals between knot values and are defined recursively in terms of lower-degree basis functions. A direct application of the Cox-DeBoor algorithm[1] may be used to evaluate the basis functions directly from the knot vector. Alternatively, an expansion of the Cox-DeBoor algorithm may be used to calculate the basis coefficients, which can then be used to calculate the basis values. The Cox-DeBoor algorithm defined on parameter u for cubic B-splines is as follows:

$$B_{i,0}(u) = \begin{cases} 1, u_i \le u < u_{i+1} \\ 0, otherwise \end{cases}$$

$$B_{i,1}(u) = \frac{u - u_i}{u_{i+1} - u_i} B_{i,0}(u) + \frac{u_{i+2} - u}{u_{i+2} - u_{i+1}} B_{i+1,0}(u)$$

$$B_{i,2}(u) = \frac{u - u_i}{u_{i+2} - u_i} B_{i,1}(u) + \frac{u_{i+3} - u}{u_{i+3} - u_{i+1}} B_{i+1,1}(u)$$

$$B_{i,3}(u) = \frac{u - u_i}{u_{i+3} - u_i} B_{i,2}(u) + \frac{u_{i+4} - u}{u_{i+4} - u_{i+1}} B_{i+1,2}(u)$$

Note that four basis values are generated for each parametric value.

The method used to evaluate basis functions varies greatly from one application to another. The method used depends on how often or how much the level of detail changes from one surface tessellation to the next. Some action must be taken with regard to the basis functions each time the level of detail changes. The simplest approach is to recalculate all of the basis functions each time the level of detail changes. There are many other approaches, but these are outside the scope of this application note. This application note makes the assumption that the basis functions have been evaluated and that the basis values are stored in an array (preferably an aligned array) in memory.

Knot vectors take the following form:

$U = [u_0, u_1, u_2, u_3, u_4, \ldots, u_{m-5}, u_{m-4}, u_{m-3}, u_{m-2}, u_{m-1}]$

where

$u_0, u_1, u_2, \ldots, u_p$ all equal 0

$u_{m-p-1}, u_{m-p}, u_{m-p+1}, \ldots, u_{m-1}$ all equal 1

$0 \le u_{p+1}, \ldots, u_{m-p-2} \le 1$

Cubic knot vectors take the form:

$U = [u_0, u_1, u_2, u_3, u_4, \ldots, u_{m-5}, u_{m-4}, u_{m-3}, u_{m-2}, u_{m-1}]$

where

$u_0 = 0$

$u_{m-1} = 1$

$0 \le u_1, u_{m-2} \le 1$

Repeating knots reduces the continuity of the surface. In fact, one of the advantages of using non-uniform parametric surfaces over using other types of parametric surfaces is having the ability to force the surface to interpolate (or pass-through) control points. The knots in other types of surfaces must be uniformly spaced (thus, they are called uniform surfaces); and consequently, cannot be repeated. This is why the continuity of uniform surfaces cannot be controlled as effectively as non-uniform surfaces. Repeating the first and last knot values four times in cubic knot vectors causes the corner control points in the control grid to be interpolated by the surface.

A *knot span* is the gap between adjacent knot values in a knot vector. Parametric values are said to 'sit' on knot spans; that is, a given parametric value will fall between two knot values in a knot vector. The implementations discussed in this application note use knot span indices to determine which set of control points should be used to evaluate $\vec{S}(u,v)$ given the current values of *u* and *v*. Figure 3 is an illustration of how knot span indices are determined.

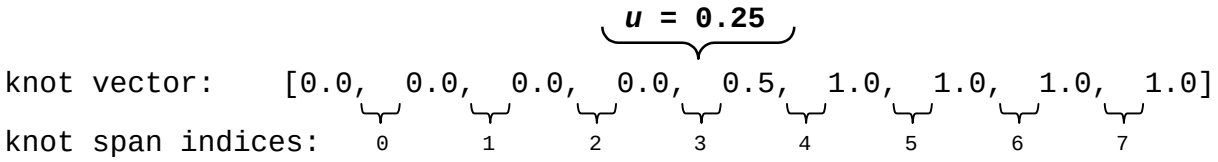

Figure 3: Knot Vector with Labeled Knot Spans. The parametric value, u = 0.25, has a knot span index equal to 3.

Appendix B - Performance Data

Performance Data Revision History

Revision	Revision History	Date
2.0	Updated with 1.2 GHz Pentium® 4 performance data	7/00
1.0	Original publication of document	9/99

The number of surface points evaluated per NURBS surface for performance profiling was 256. The reason 256 points were chosen is because the per-surface point performance reaches a steady state at that point. Evaluating more than 256 surface points per surface caused some performance variations. Evaluating less than 256 surface points caused operating system overhead to interfere with performance measurements.

The control-point grid size was arbitrarily chosen to be 10-by-10. There were 10 control points in the *u* direction and 10 control points in the *v* direction. This control grid size was chosen arbitrarily. Changing the control grid size results in performance that differs from the results presented here. Increasing the number of control points in the control grid causes performance to degrade due to an increase in the required number of memory accesses.

Table 1: Performance Data of NURBS Surface Tessellation.

Average elapsed microseconds per surface point evaluated.

Performance Data in Microseconds/surface point		
Implementation	**Pentium III Processor (733 MHz)**	**Pentium 4 Processor (1.2 GHz)**
Intel® x87 Architecture	0.377	0.263
cubic Intel x87 Architecture	0.344	0.227
unrolled cubic Intel x87 Architecture	0.356	0.192
Streaming SIMD Extensions (SSE)	0.306	0.190
cubic SSE	0.161	0.0967

Table 2: Speedups from Table 1 Performance Data.

Speed-ups are calculated by dividing the values from the second column of Table 1 by the values from the third column of Table 1.

Implementation	Pentium 4 Processor Speedup
Intel x87 Architecture	1.43
cubic Intel x87 Architecture	1.52
unrolled cubic Intel x87 Architecture	1.86
SSE	1.61
cubic SSE	1.67
average speed-up	**1.62**

Performance was measured assuming a regular caching environment (some other application notes may use a "perfect cache" environment). Performance was measured using a 733 MHz Pentium III processor and a 1.2 GHz Pentium 4 processor. See Test Systems Configuration on page B-3 (Tables 3 and 4) for a detailed description of both systems.

Note that software prefetching can be used to hide the performance impact of latency for smaller numbers of control points. Software prefetching is not used in any of the code used in the development of this application note.

The fastest solution is the SSE cubic implementation on Pentium 4. The best Pentium 4 processor speed-up is seen in the unrolled cubic x87 Architecture implementation at 1.86x. The second-best speed-up is seen in the cubic SSE implementation at 1.67x.

On the Pentium III processor, the fastest SSE implementation is 2.14x faster than the fastest Intel x87 Architecture implementation. On the Pentium 4 processor, the fastest SSE implementation is 1.98x faster than the fastest Intel x87 Architecture implementation. The fastest SSE implementation on the Pentium 4 processor is 3.56x faster than the fastest Intel x87 Architecture implementation on the Pentium III processor.

For the most part, performance scales close to processor frequency between the Pentium III processor and the Pentium 4 processor. NURBS surface tessellation is greatly improved by the use of SSE and by the Pentium 4 processor.

Test Systems Configuration

Table 3: Pentium III Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 4: Pentium 4 Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195